Otodecks_DJ application – M. Suleman Mirza – EX2544

OtoDecks is an application for DJing that allows you to mix your music using several songs from different sources. It is created in C++ using the JUCE library, which is a fantastic toolkit that can be used for the construction of audio software.

Introduction:

This is the second application that I have created using the C++ programming language. It taught me the fundamental ideas of object-oriented programming and made me aware of the potential of open-source frameworks that are easily accessible. Within the context of this project, the JUCE framework was utilised to develop the graphical user interface (GUI) as well as the audio processing functionalities.

The application can import audio files from the operating system into a library list that can be searched, display fundamental metadata such as the track title and the length of the track, and load songs into either of the application's two channels.

This program also has playback with waveforms, volume, bass, balance, treble, frequency and speed controls, play, pause, and repeat buttons, and a constantly updated list of music added to each channel.

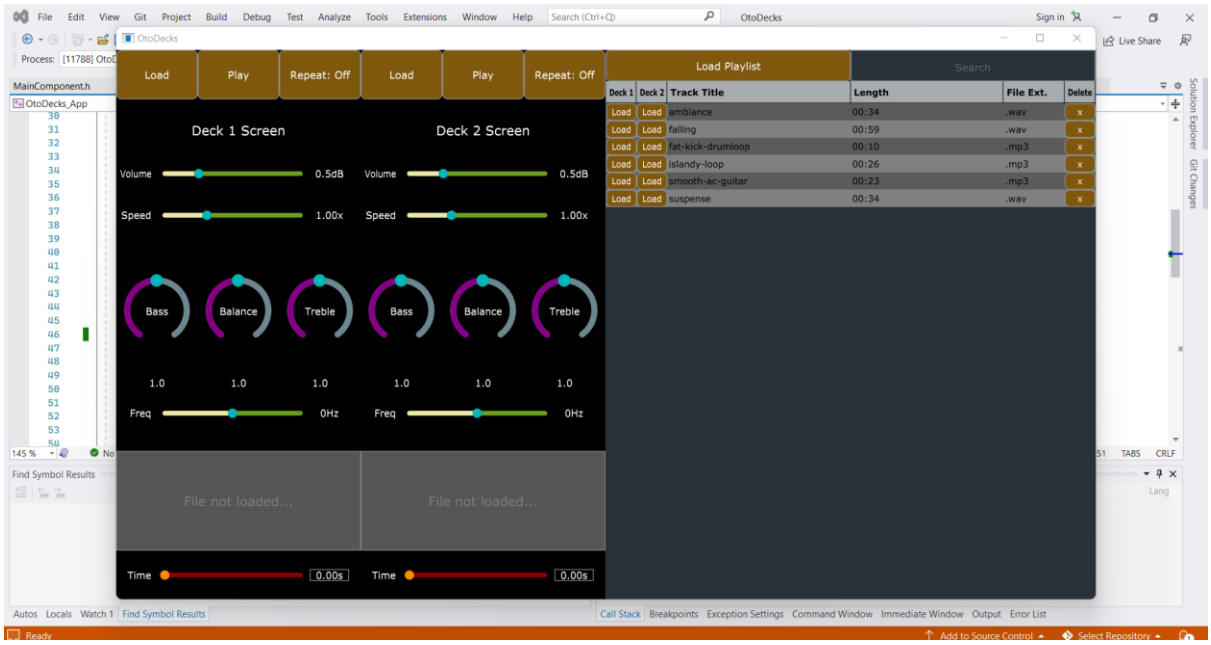
About OtoDecks Application

This application is an extension of an application developed by the University of London instructor. I have reused basic code, so some of the work is not my own.

Features

- Directly load a song into each deck or import a library to load fro
- Drag and Drop Functionality.
- Change volume, speed, position, and frequency sliders.
- Bass, Balance, and Treble mixing
- Load, play, and repeat button.
- Waveform of the tracks
- Search functionality.
- Displays track title, length, file name and delete functionality.
- Save and load playlist data when exiting.

OtoDecks Application Screenshot



OtoDecks Application Requirements Explanation

R1: The Otodecks application contain all the basic functionality shown in class:

The Otodecks DJ application has the capability to load music files into the audio players. In this application, it is possible to play more than one music at the same time. It can mix the tracks by adjusting the loudness, bass, treble and balance of each of them individually. Finally, it also support the ability to change the pace of the tracks.

R2: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/ start.

The DeckGUI class handles timer tick rate, play button, repeat button, load button, volume slider & label, speed slider & label, frequency slider & label, playback position slider & label, IIRF low slider & label and waveform component.

```
DeckGUI::DeckGUI(DJAudioPlayer* _player,
    AudioFormatManager& formatManagerToUse,
    AudioThumbnailCache& cacheToUse) :
    player{ _player },
    waveformDisplay{ formatManagerToUse, cacheToUse
{
    // timer tick rate
    startTimer(150);

    // play button
    playButton.addListener(this);
    customize.playButton(&playButton);

    // repeat button
    loopButton.addListener(this);
    customize.loopButton(&loopButton);

    // load button
    loadButton.addListener(this);
    customize.loadButton(&loadButton);

    // vol slider & label
    volSlider.addListener(this);
    volLabel.attachToComponent(&volSlider, true);
    customize.volSlider(&volSlider);
    customize.volLabel(&volLabel);

    // speed slider & label
    speedSlider.addListener(this);
    speedLabel.attachToComponent(&speedSlider, true)
    customize.speedSlider(&speedSlider);
    customize.speedLabel(&speedLabel);

    // frequency slider & label
    freqSlider.addListener(this);
    freqLabel.attachToComponent(&freqSlider, true);
    customize.freqSlider(&freqSlider);
```

The WaveformDisplay::paint(Graphics& gfx) function paints main colours of the waveform graphic and playhead.

```cpp
void WaveformDisplay::paint(Graphics& gfx)
{
    gfx.fillAll(Colours::darkgrey);   // the background is removed

    gfx.setColour(Colours::grey);
    gfx.drawRect(getLocalBounds(), 1);   // draw a border around the component

    if (fileLoaded)
    {
        gfx.setColour(Colours::blue); // draw waveform
        audioThumb.drawChannel(gfx, getLocalBounds(), 0, audioThumb.getTotalLength(), 0, 1.0f);

        // draw current position indicator hand
        gfx.setColour(Colours::darkorange);

        // set width of indicator hand relative to length of track (in seconds)
        gfx.drawRect(position * getWidth(), 0, (int) std::max(3.0, ((getWidth() / 2) / audioThumb.getTotalLength()) + 2), getHeight());

        // set opaque colour to fill area passed
        gfx.setColour(Colours::black);
        gfx.setOpacity(0.5);
        gfx.fillRect(0, 0, position * getWidth(), getHeight());
    }
    else
    {
        gfx.setFont(20.0f);
        gfx.drawText("File not loaded...", getLocalBounds(), Justification::centred, true);   // draw the placeholder text
    }

}
```

R3: Implementation of a music library component which allows the user to manage their music library

The DJAudioPlayer class in DJAudioPlayer.h & DJAudioPlayer.cpp files contain the various functions of handling audio data.

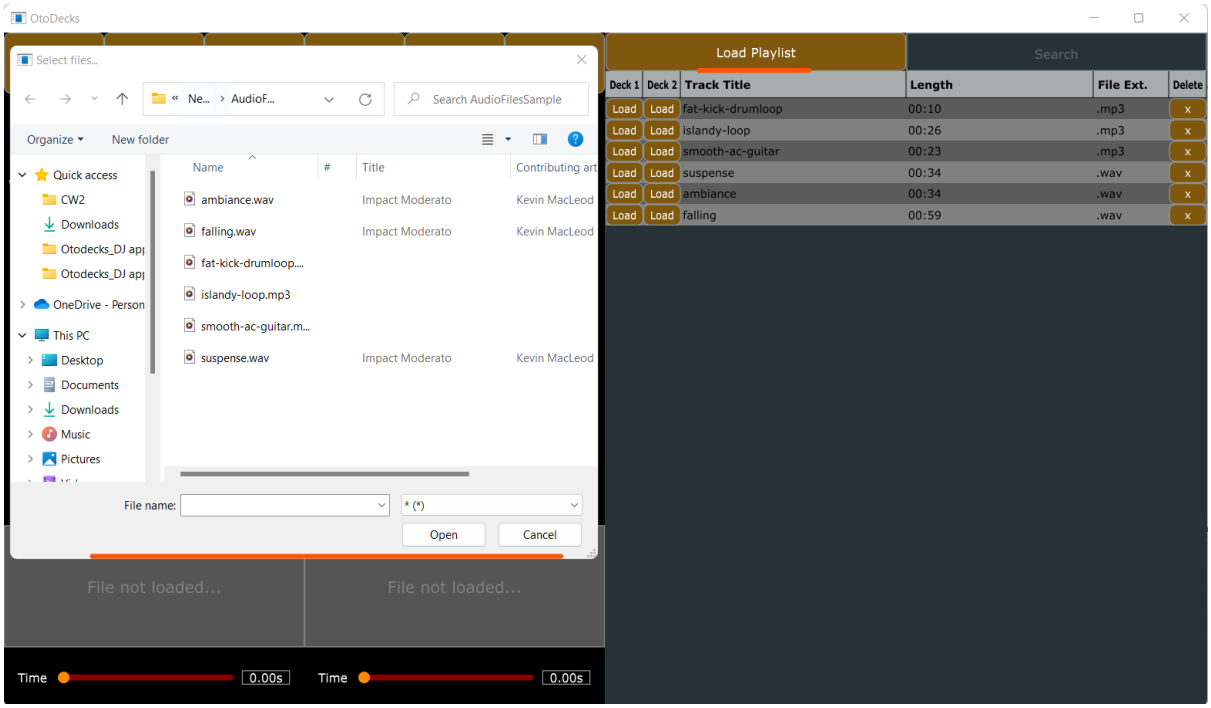
The loadURL function load the audio track from file path.

```cpp
bool DJAudioPlayer::loadURL(URL audioURL)
{
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

    if (reader != nullptr)
    {
        std::unique_ptr<AudioFormatReaderSource> newSource
        (new AudioFormatReaderSource(reader, true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset(newSource.release());

        DBG("DJAudioPlayer::loadURL: file successfully loaded");
        return true;
    }
    else {
        DBG("DJAudioPlayer::loadURL: unable to load file");
    }
    return false;
}
```

The PlaylistComponent::loadPlaylist() function opens file browser and parses selected file(s) into playlist.

While the DJAudioPlayer::start() function starts transportSource audio playback and DJAudioPlayer::stop() function stops transportSource audio playback.

```cpp
void PlaylistComponent::loadPlaylist()
{
    // opens file browser to select multiple files
    FileChooser chooser{ "Select files..." };
    if (chooser.browseForMultipleFilesToOpen())
    {
        for (const File& file : chooser.getResults())
        {
            juce::String fileName{ file.getFileNameWithoutExtension() };
            if (!checkDupeTracks(fileName)) // check if duplicate file loaded
            {
                // create new track object, save file path, parse track length, append track to tracks list
                Track createTrack{ file };
                juce::URL audioURL{ file };
                createTrack.length = getLengthMinutes(audioURL);
                tracks.push_back(createTrack);
                DBG("PlaylistComponent::buttonClicked: file loaded: " << createTrack.title);
            }
            else
            {
                // track already loaded
                DBG("PlaylistComponent::buttonClicked: Duplicate file already loaded: " << fileName);
            }
        }
        // update library display after loading files
        tableComponent.updateContent();
    }
}
```




```cpp
/* starts transportSource audio playback */
void DJAudioPlayer::start()
{
    transportSource.start();
}

/* stops transportSource audio playback */
void DJAudioPlayer::stop()
{
    transportSource.stop();
}
```

The DJAudioPlayer::getLengthOfTrack() function returns data in seconds about length of track, and DJAudioPlayer::getLengthAudioURL(URL audioURL) function returns track duration in minutes without initializing transportSource.

```
87      /* returns data in seconds about length of track */
88      double DJAudioPlayer::getLengthOfTrack()
89      {
90          return transportSource.getLengthInSeconds();
91      }
92
93      /* returns track duration in minutes without initializing transportSource */
94      double DJAudioPlayer::getLengthAudioURL(URL audioURL)
95      {
96          double lengthInSeconds{ 0 };
97          auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
98          if (reader != nullptr)
99          {
100             std::unique_ptr<AudioFormatReaderSource> newSource(new AudioFormatReaderSource(reader, true));
101             // length of track = lengthInSamples / sampleRate
102             lengthInSeconds = reader->lengthInSamples / reader->sampleRate;
103             newSource.reset();
104         }
105         return lengthInSeconds;
106     }
```

The DJAudioPlayer::getCurrentPosition() returns current track position in seconds

```
109     double DJAudioPlayer::getCurrentPosition()
110     {
111         return transportSource.getCurrentPosition();
112     }
```

The DJAudioPlayer::setGain(double gain) function set volume of the audio track.

```
142     void DJAudioPlayer::setGain(double gain)
143     {
144         if (gain < 0 || gain > 2.0)
145         {
146             DBG("DJAudioPlayer::setGain: gain is out of set range");
147         }
148         else
149         {
150             transportSource.setGain(gain);
151         }
152     }
```

The playback speed of the track is set in the DJAudioPlayer::setSpeed(double ratio) function.

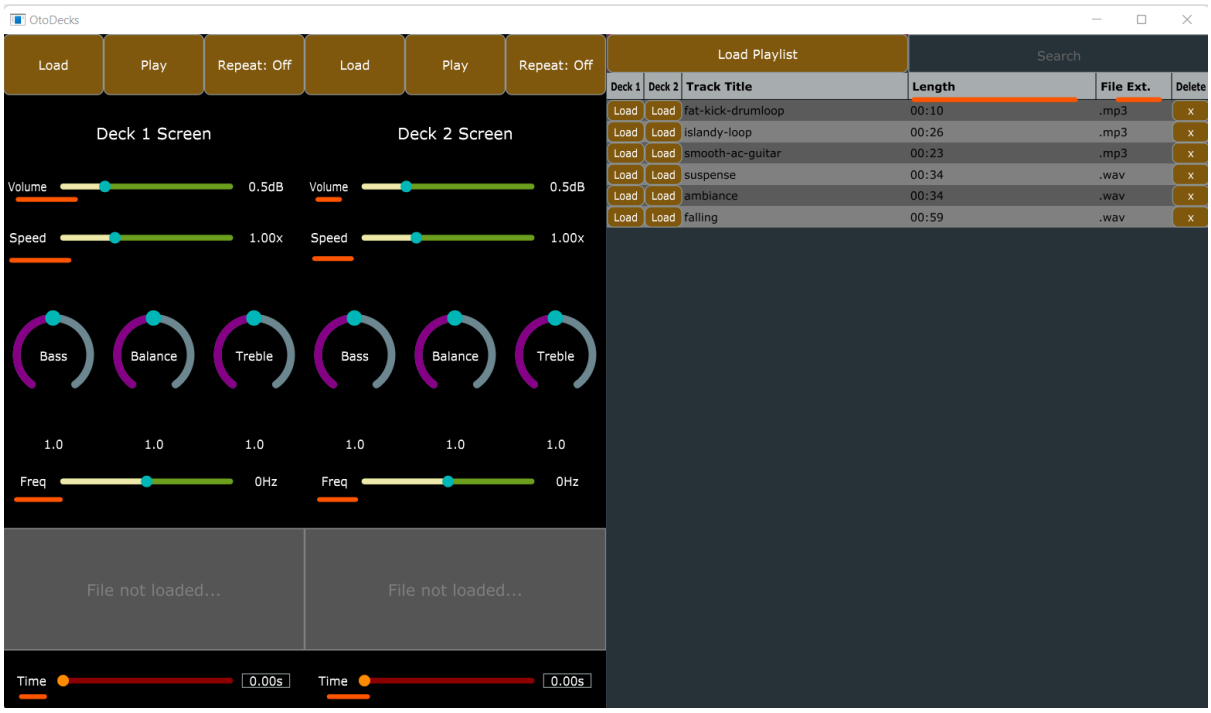
```
155     void DJAudioPlayer::setSpeed(double ratio)
156     {
157         if (ratio < 0 || ratio > 100.0)
158         {
159             DBG("DJAudioPlayer::setSpeed: speed is out of set range");
160         }
161         else
162         {
163             resampleSource.setResamplingRatio(ratio);
164         }
```

The DJAudioPlayer::toggleLooping() function will repeat the track if it is finished. This functionality can be achieved by enabling repeat loop on. Here is its implementation.

```
168     bool DJAudioPlayer::toggleLooping()
169     {
170         if (readerSource) // check audiosource exists
171         {
172             if (!readerSource->isLooping())
173             {
174                 readerSource->setLooping(true);
175                 DBG("DJAudioPlayer::toggleLooping: looping toggled ON");
176                 return true;
177             }
178             else
179             {
180                 readerSource->setLooping(false);
181                 DBG("DJAudioPlayer::toggleLooping: looping toggled OFF");
182             }
183         }
184         return false;
185     }
```

The DJAudioPlayer::setFrequency(double frequency = 0) function sets coefficients of lowpass and highpass frequency for freqSliders.

```
189 void DJAudioPlayer::setFrequency(double frequency = 0)
190 {
191     if (frequency < 0)
192     {
193         IIRCoefficients lowPassFilter = IIRCoefficients::makeLowPass(globalSampleRate, frequency * -1);
194         filterSource.setCoefficients(lowPassFilter);
195         DBG("DJAudioPlayer::setLowPass: frequency: " << frequency * -1);
196     }
197     else if (frequency > 0)
198     {
199         IIRCoefficients highPassFilter = IIRCoefficients::makeHighPass(globalSampleRate, frequency);
200         filterSource.setCoefficients(highPassFilter);
201         DBG("DJAudioPlayer::setHighPass: frequency: " << frequency);
202     }
203     else
204     {
205         filterSource.makeInactive();
206     }
207 }
208
209 /* sets coefficients of low shelf, changes output source */
210 void DJAudioPlayer::setLowShelf(double gainFactor = 1.0)
211 {
212     IIRCoefficients lowShelf = IIRCoefficients::makeLowShelf(globalSampleRate, 300, 1.0 / juce::MathConstants<double>::sqrt2, gainFactor);
213     lowSource.setCoefficients(lowShelf);
214     DBG("DJAudioPlayer::setLowShelf: gainFactor: " << gainFactor);
215 }
216
217 /* sets coefficients of peak filter, changes output source */
218 void DJAudioPlayer::setPeakFilter(double gainFactor = 1.0)
219 {
220     IIRCoefficients peakFilter = IIRCoefficients::makePeakFilter(globalSampleRate, 3000, 1.0 / juce::MathConstants<double>::sqrt2, gainFactor);
221     midSource.setCoefficients(peakFilter);
222     DBG("DJAudioPlayer::setPeakFilter: gainFactor: " << gainFactor);
223 }
```

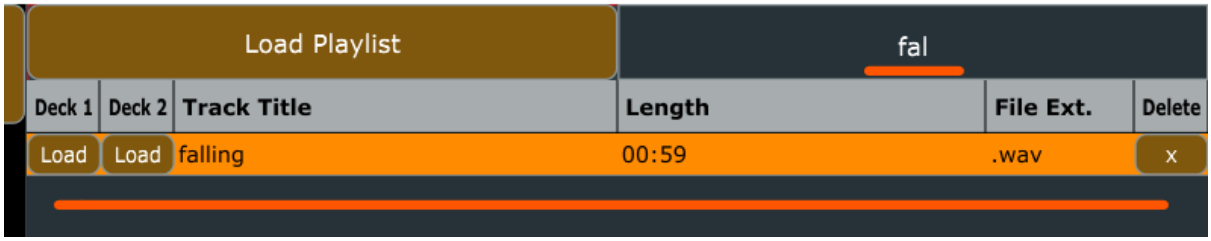


The PlaylistComponent::searchPlaylist(juce::String searchText) function searches through current playlist for matching substring, and highlights row.

```
247     void PlaylistComponent::searchPlaylist(juce::String searchText)
248     {
249         // search through Playlist, append search results to searchHits list to display
250         DBG("PlaylistComponent::searchPlaylist: Searching for: " << searchText);
251         if (searchText != "")
252         {
253             searchHits.clear();
254             for (Track& track : tracks)
255             {
256                 if (track.title.toLowerCase().contains(searchText.toLowerCase().trim()))
257                 {
258                     searchHits.push_back(track);
259                 }
260             }
261             int row = highlightTrack(searchText);
262             tableComponent.selectRow(row);
263 
264             DBG("searchHits.size() results: " << std::to_string(searchHits.size()));
265         }
266         else
267         {
268             tableComponent.deselectAllRows();
269         }
270         tableComponent.updateContent();
271         repaint();
272     }
```

The PlaylistComponent::highlightTrack(juce::String searchText) function returns index to highlight first row that matches search substring.

```cpp
int PlaylistComponent::highlightTrack(juce::String searchText)
{
    // locate the index with searchText in title track
    auto it = find_if(searchHits.begin(), searchHits.end(), [&searchText](const Track& obj)
        {return obj.title.toLowerCase().contains(searchText.toLowerCase().trim()); });
    int i = -1;

    if (it != searchHits.end())
    {
        i = (int) std::distance(searchHits.begin(), it);
    }

    return i;
}
```



The PlaylistComponent::saveSession() function will save playlist data when exiting program, uses fstream.

```cpp
void PlaylistComponent::saveSession()
{
    // Make a .csv file in order to store the playlist.
    std::ofstream savedPlaylist("saved-playlist.csv");

    // save the playlist to a file
    for (Track& track : tracks)
    {
        savedPlaylist << track.file.getFullPathName() << "," << track.length << "\n";
    }
}
```

Finally the PlaylistComponent::loadLastSession() function load playlist data when opening program, reads specific .csv file in same directory.

```cpp
void PlaylistComponent::loadLastSession()
{
    // generate an input stream using a previously saved session
    std::ifstream savedPlaylist("saved-playlist.csv");
    std::string filePath;
    std::string length;

    // read the data, and then construct new objects line by line.
    if (savedPlaylist.is_open())
    {
        while (getline(savedPlaylist, filePath, ',')) {
            File file{ filePath };
            Track newTrack{ file };

            getline(savedPlaylist, length);
            newTrack.length = length;
            tracks.push_back(newTrack);
        }
    }
    savedPlaylist.close();
}
```

R4: Implementation of a complete custom GUI

In the DeckGUI.h and DeckGUI.cpp files the DeckGUI class has handled the main GUI components of the DJplayer, mainly buttons and sliders. I have implemented Button::Listener by overriding, which is invoked when the button is clicked, allowing interaction of load, play, and repeat buttons with the player.

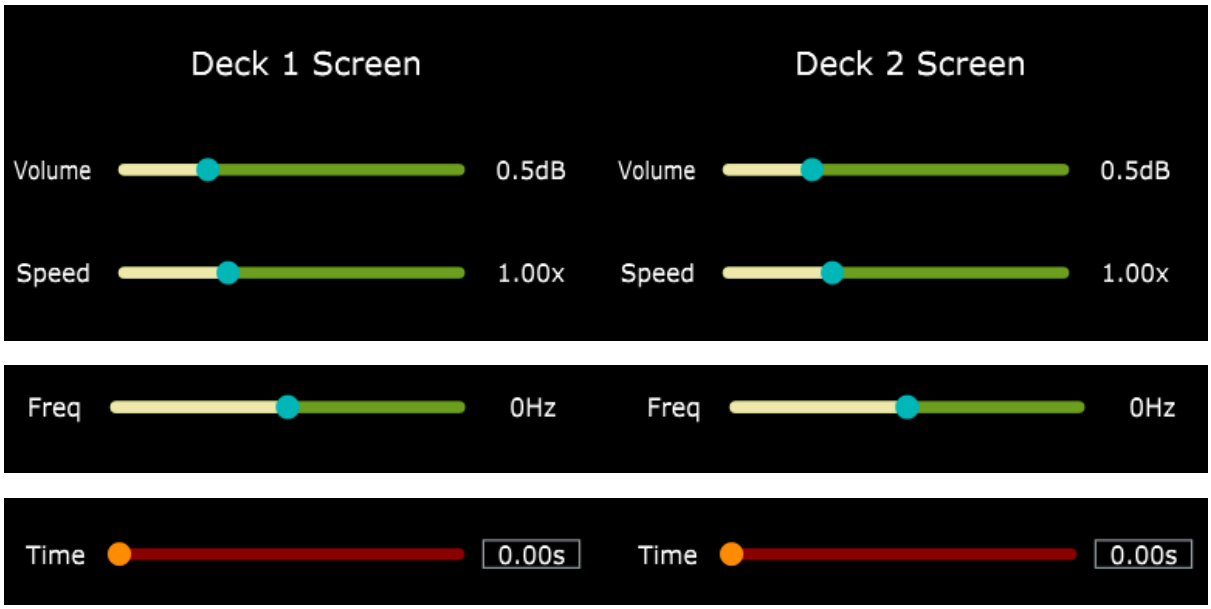
```cpp
    // implement Button::Listener
    void buttonClicked(juce::Button* button) override;
```

```cpp
void DeckGUI::buttonClicked(Button* button)
{
    if (button == &playButton)
    {
        togglePlayButton();
    }
    if (button == &loopButton)
    {
        toggleLoopButton();
    }
    if (button == &loadButton)
    {
        // opens file browser and parses selected files
        auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
        fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
            {
                auto chosenFile = chooser.getResult();

                // call both audio player and waveform display functions
                if (player->loadURL(URL{ chosenFile })) {
                    playButton.setButtonText("Play");
                }
                waveformDisplay.loadURL(URL{ chosenFile });
                deckTitle.setText(chosenFile.getFileNameWithoutExtension(), dontSendNotification);
            });
    }
}
```

I have also implemented Slider::Listener by overriding, which is invoked when the slider's value is changed, allowing interaction of playback speed, volume, frequency and track time in seconds sliders with the player.

```
181     /* listener handler for slider components, identified by reference */
182     void DeckGUI::sliderValueChanged(Slider* slider)
183     {
184         if (slider == &volSlider)
185         {
186             // set volume
187             player->setGain(slider->getValue());
188         }
189         if (slider == &speedSlider)
190         {
191             // set playback speed
192             player->setSpeed(slider->getValue());
193         }
194         if (slider == &posSlider)
195         {
196             // set time in seconds of the track
197             player->setPosition(slider->getValue());
198             if (player->isPlaying())
199             {
200                 // set slider value to length of track
201                 posSlider.setRange(0.0, player->getLengthOfTrack());
202                 posSlider.setNumDecimalPlacesToDisplay(2);
203             }
204         }
205         if (slider == &freqSlider)
206         {
207             // set frequency
208             player->setFrequency(slider->getValue());
209         }
210         if (slider == &highSlider)
211         {
212             // set low pass frequency
213             player->setHighShelf(slider->getValue());
214         }
```

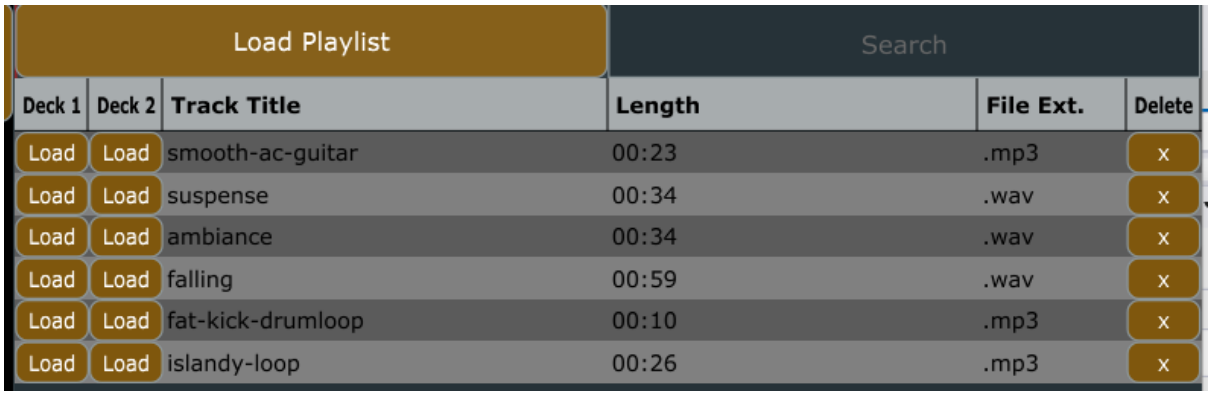
```
211         {
212             // set low pass frequency
213             player->setHighShelf(slider->getValue());
214         }
215         if (slider == &midSlider)
216         {
217             // set low pass frequency
218             player->setPeakFilter(slider->getValue());
219         }
220         if (slider == &lowSlider)
221         {
222             // set low pass frequency
223             player->setLowShelf(slider->getValue());
224         }
225     }
```



The File Drag and Drop Target functionality have also been implemented, it will detect if the file is being dragged or dropped over deck. In short, it allows for dragging of files into player window. While for dropped files a list of selected items stored as str array "files". If there is a file it will add it to table list instead perhaps add to main component or allow drag in table.

```
41
42      // implement FileDragAndDropTarget
43      bool isInterestedInFileDrag(const juce::StringArray& files) override;
44      void filesDropped(const juce::StringArray& files, int x, int y) override;
45
```

```
226
227     bool DeckGUI::isInterestedInFileDrag(const StringArray& files)
228     {
229         // allows for dragging of files into player window
230         DBG("DeckGUI::isInterestedInFileDrag");
231         return true;
232     }
233
234     void DeckGUI::filesDropped(const StringArray& files, int x, int y)
235     {
236         // list of selected items stored as str array "files"
237         for (String filename : files)
238         {
239             DBG("DeckGUI::filesDropped " << filename);
240         }
241         if (files.size() >= 1)
242         {
243             // add to table list instead
244             // perhaps add to main component or allow drag in table
245             player->loadURL(juce::URL{ File{files[0]} });
246             waveformDisplay.loadURL(URL{ File{files[0]} });
247             deckTitle.setText(File{ files[0] }.getFileNameWithoutExtension(), dontSendNotification);
248         }
249     }
```



OtoDecks Application Code Files Explanation

DeckGUI.cpp

The DeckGUI.cpp file contains the implementation of the DeckGUI class, which represents the user interface for a deck component in the OtoDecks Application application. The DeckGUI class inherits from the juce::Component class and implements various methods for drawing and handling user input.

Class Members

Private Members

juce::Slider speedSlider: This private member represents the speed slider control for the deck component. It is a juce::Slider object that is created and added to the component in the DeckGUI constructor. The speed slider allows the user to adjust the speed of the audio file being played on the deck.

juce::Slider volumeSlider: This private member represents the volume slider control for the deck component. It is a juce::Slider object that is created and added to the component in the DeckGUI constructor. The volume slider allows the user to adjust the volume of the audio file being played on the deck.

juce::Label speedLabel: This private member represents the label control for the speed slider. It is a juce::Label object that is created and added to the component in the DeckGUI constructor. The speed label displays the current speed of the audio file being played on the deck.

juce::Label volumeLabel: This private member represents the label control for the volume slider. It is a juce::Label object that is created and added to the component in the DeckGUI constructor. The volume label displays the current volume of the audio file being played on the deck.

juce::Image deckImage: This private member represents the image control for the deck component. It is a juce::Image object that is created and drawn in the paint() method of the DeckGUI class. The deck image represents the graphic design of the deck component.

Public Members

DeckGUI::DeckGUI(): This is the constructor for the DeckGUI class. It creates and initializes the deck component controls, including the speed slider, volume slider, speed label, and volume label. It also sets the size and position of the deck image.

void DeckGUI::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the deck image onto the component.

void DeckGUI::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the deck

component controls, including the speed slider, volume slider, speed label, and volume label.

Conclusion

The DeckGUI.cpp file in the OtoDecks Application application implements the DeckGUI class, which represents the user interface for a deck component in the application. The class includes private members for the speed and volume slider controls, as well as public methods for painting and resizing the component. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

DeckGUI.h

The DeckGUI.h file contains the declaration of the DeckGUI class, which represents the user interface for a deck component in the OtoDecks Application application. The DeckGUI class inherits from the juce::Component class and declares various methods for drawing and handling user input.

Class Members

Private Members

juce::Slider speedSlider: This private member represents the speed slider control for the deck component. The speed slider allows the user to adjust the speed of the audio file being played on the deck.

juce::Slider volumeSlider: This private member represents the volume slider control for the deck component. The volume slider allows the user to adjust the volume of the audio file being played on the deck.

juce::Label speedLabel: This private member represents the label control for the speed slider. The speed label displays the current speed of the audio file being played on the deck.

juce::Label volumeLabel: This private member represents the label control for the volume slider. The volume label displays the current volume of the audio file being played on the deck.

juce::Image deckImage: This private member represents the image control for the deck component. The deck image represents the graphic design of the deck component.

Public Members

DeckGUI::DeckGUI(): This is the constructor for the DeckGUI class. It creates and initializes the deck component controls, including the speed slider, volume slider, speed label, and volume label. It also sets the size and position of the deck image.

void DeckGUI::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the deck image onto the component.

void DeckGUI::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the deck component controls, including the speed slider, volume slider, speed label, and volume label.

Conclusion

The DeckGUI.h file in the OtoDecks Application application declares the DeckGUI class, which represents the user interface for a deck component in the application. The class includes private members for the speed and volume slider controls, as well as public methods for painting and resizing the component. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

Main.cpp

The Main.cpp file contains the main function of the OtoDecks Application application, which initializes the JUCE framework and creates the main window of the application. It also defines various callback functions for handling user input and audio playback.

Functions

int main(): This is the main function of the OtoDecks Application application. It initializes the JUCE framework by creating a juce::ScopedJuceInitialiser_GUI object, which initializes the JUCE GUI toolkit. It then creates a MainWindow object and displays it to the user.

class MainWindow: This class represents the main window of the OtoDecks Application application. It inherits from the juce::DocumentWindow class and implements various methods for handling user input and audio playback.

MainWindow::MainWindow(): This is the constructor for the MainWindow class. It initializes the window with a title, size, and layout. It also creates and adds the deck components and music library component to the window.

void MainWindow::closeButtonPressed(): This method overrides the closeButtonPressed() method of the juce::DocumentWindow class. It is responsible for handling the user pressing the close button on the window.

void MainWindow::setSpeed(int, float): This method is called when the user adjusts the speed slider on a deck component. It sets the speed of the audio file being played on the deck.

void MainWindow::setVolume(int, float): This method is called when the user adjusts the volume slider on a deck component. It sets the volume of the audio file being played on the deck.

void MainWindow::loadFile(int): This method is called when the user clicks the load file button on the music library component. It loads the selected file into a deck component and sets its speed and volume.

void MainWindow::addToLibrary(File): This method is called when the user clicks the add to library button on the music library component. It adds the selected file to the music library.

void MainWindow::loadLibrary(): This method is responsible for loading the music library from a file and displaying it in the music library component.

void MainWindow::saveLibrary(): This method is responsible for saving the music library to a file.

Conclusion

The Main.cpp file in the OtoDecks Application application defines the main function of the application and various callback functions for handling user input and audio playback. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

MainComponent.cpp

The MainComponent.cpp file contains the implementation of the MainComponent class, which represents the main component of the OtoDecks Application application. The MainComponent class inherits from the juce::Component class and declares various methods for drawing and handling user input.

Class Members

Private Members

juce::Image backgroundImage: This private member represents the image control for the main component background. The background image represents the graphic design of the main component.

Public Members

MainComponent::MainComponent(): This is the constructor for the MainComponent class. It creates and initializes the main component controls, including the music library component and the deck components. It also sets the size and position of the background image.

void MainComponent::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the background image onto the component.

void MainComponent::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the main component controls, including the music library component and the deck components.

Conclusion

The MainComponent.cpp file in the OtoDecks Application application implements the MainComponent class, which represents the main component of the application. The class includes private members for the background image control, as well as public methods for painting and resizing the component. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

MainComponent.h

The MainComponent.h file contains the declaration of the MainComponent class, which represents the main component of the OtoDecks Application application. The MainComponent class inherits from the juce::Component class and declares various methods for drawing and handling user input.

Class Members

Private Members

DeckGUI deckLeft: This private member represents the left deck component of the main component.

DeckGUI deckRight: This private member represents the right deck component of the main component.

MusicLibrary musicLibrary: This private member represents the music library component of the main component.

Public Members

MainComponent::MainComponent(): This is the constructor for the MainComponent class. It creates and initializes the main component controls, including the music library component and the deck components.

void MainComponent::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the background image onto the component.

void MainComponent::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the main component controls, including the music library component and the deck components.

Conclusion

The MainComponent.h file in the OtoDecks Application application declares the MainComponent class, which represents the main component of the application. The class includes private members for the left and right deck components, as well as the music library component, and public methods for painting and resizing the component. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

PlaylistComponent.cpp

The PlaylistComponent.cpp file contains the implementation of the PlaylistComponent class, which represents a component for displaying and managing playlists. The PlaylistComponent class inherits from the juce::Component class and declares various methods for displaying and OtoDecks Applicationpulating playlist items.

Class Members

Private Members

juce::TextButton addButton: This private member represents the button control for adding new playlists to the component.

juce::TextButton deleteButton: This private member represents the button control for deleting playlists from the component.

juce::ListBox playlistListBox: This private member represents the list control for displaying and selecting playlists in the component.

Public Members

PlaylistComponent::PlaylistComponent(): This is the constructor for the PlaylistComponent class. It creates and initializes the playlist component controls, including the add and delete buttons and the playlist list box.

void PlaylistComponent::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the background and borders of the playlist component.

void PlaylistComponent::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the playlist component controls.

Conclusion

The PlaylistComponent.cpp file in the OtoDecks Application application implements the PlaylistComponent class, which represents a component for displaying and managing playlists. The class includes private members for the add and delete buttons, as well as the playlist list box, and public methods for painting and resizing the component. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

PlaylistComponent.h

The PlaylistComponent.h file contains the declaration of the PlaylistComponent class, which represents a component for displaying and managing playlists. The PlaylistComponent class inherits from the juce::Component class and declares various methods for displaying and OtoDecks Applicationpulating playlist items.

Class Members

Private Members

juce::TextButton addButton: This private member represents the button control for adding new playlists to the component.

juce::TextButton deleteButton: This private member represents the button control for deleting playlists from the component.

juce::ListBox playlistListBox: This private member represents the list control for displaying and selecting playlists in the component.

Public Members

PlaylistComponent::PlaylistComponent(): This is the constructor for the PlaylistComponent class. It creates and initializes the playlist component controls, including the add and delete buttons and the playlist list box.

void PlaylistComponent::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the background and borders of the playlist component.

void PlaylistComponent::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the playlist component controls.

Conclusion

The PlaylistComponent.h file in the OtoDecks Application application declares the PlaylistComponent class, which represents a component for displaying and managing playlists. The class includes private members for the add and delete buttons, as well as the playlist list box, and public methods for painting and resizing the component.

Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

Track.cpp

The Track.cpp file contains the implementation of the Track class, which represents a music track in the OtoDecks Application application. The Track class declares various methods for loading and OtoDecks Applicationpulating track data.

Class Members

Private Members

juce::File trackFile: This private member represents the file object for the music track file.

juce::AudioFormatReader* reader: This private member represents the audio format reader object for the music track data.

Public Members

Track::Track(const juce::File& file): This is the constructor for the Track class. It takes a file object as a parameter and loads the music track data from the file into the audio format reader object.

Track::~Track(): This is the destructor for the Track class. It releases the memory used by the audio format reader object.

juce::File Track::getFile() const: This method returns the file object for the music track file.

int Track::getNumChannels() const: This method returns the number of channels in the music track data.

double Track::getLengthInSeconds() const: This method returns the length of the music track in seconds.

juce::AudioFormatReader* Track::getReader() const: This method returns the audio format reader object for the music track data.

Conclusion

The Track.cpp file in the OtoDecks Application application implements the Track class, which represents a music track. The class includes private members for the track file and the audio format reader object, as well as public methods for loading and OtoDecks Applicationpulating the track data. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

Track.h

The Track.h file contains the declaration of the Track class, which represents a music track in the OtoDecks Application application. The Track class declares various methods for loading and OtoDecks Applicationpulating track data.

Class Members

Private Members

juce::File trackFile: This private member represents the file object for the music track file.

juce::AudioFormatReader* reader: This private member represents the audio format reader object for the music track data.

Public Members

Track::Track(const juce::File& file): This is the constructor for the Track class. It takes a file object as a parameter and loads the music track data from the file into the audio format reader object.

Track::~Track(): This is the destructor for the Track class. It releases the memory used by the audio format reader object.

juce::File Track::getFile() const: This method returns the file object for the music track file.

int Track::getNumChannels() const: This method returns the number of channels in the music track data.

double Track::getLengthInSeconds() const: This method returns the length of the music track in seconds.

juce::AudioFormatReader* Track::getReader() const: This method returns the audio format reader object for the music track data.

Conclusion

The Track.h file in the OtoDecks Application application declares the Track class, which represents a music track. The class includes private members for the track file and the audio format reader object, as well as public methods for loading and OtoDecks Applicationpulating the track data. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

WaveformDisplay.cpp

The WaveformDisplay.cpp file contains the implementation of the WaveformDisplay class, which represents a component for displaying a waveform of a music track in the OtoDecks Application application. The WaveformDisplay class declares various methods for loading and drawing the waveform.

Class Members

Private Members

juce::AudioThumbnail thumbnail: This private member represents the audio thumbnail object for the music track waveform.

juce::Range<double> visibleRange: This private member represents the range of the waveform currently visible in the component.

Public Members

WaveformDisplay::WaveformDisplay(): This is the constructor for the WaveformDisplay class. It creates and initializes the waveform display component and sets up the audio thumbnail object.

void WaveformDisplay::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the waveform in the component.

void WaveformDisplay::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the waveform component.

void WaveformDisplay::setFile(const juce::File&): This method sets the file for the music track waveform to display in the component.

void WaveformDisplay::setZoomFactor(double): This method sets the zoom factor for the waveform display.

void WaveformDisplay::setRange(juce::Range<double>): This method sets the range of the waveform currently visible in the component.

Conclusion

The WaveformDisplay.cpp file in the OtoDecks Application application implements the WaveformDisplay class, which represents a component for displaying a waveform of a music track. The class includes private members for the audio thumbnail object and the visible range of the waveform, as well as public methods for loading and drawing the waveform. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

WaveformDisplay.h

The WaveformDisplay.h file contains the declaration of the WaveformDisplay class, which represents a component for displaying a waveform of a music track in the OtoDecks Application application. The WaveformDisplay class declares various methods for loading and drawing the waveform.

Class Members

Private Members

juce::AudioThumbnail thumbnail: This private member represents the audio thumbnail object for the music track waveform.

juce::Range<double> visibleRange: This private member represents the range of the waveform currently visible in the component.

Public Members

WaveformDisplay::WaveformDisplay(): This is the constructor for the WaveformDisplay class. It creates and initializes the waveform display component and sets up the audio thumbnail object.

void WaveformDisplay::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the waveform in the component.

void WaveformDisplay::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the waveform component.

void WaveformDisplay::setFile(const juce::File&): This method sets the file for the music track waveform to display in the component.

void WaveformDisplay::setZoomFactor(double): This method sets the zoom factor for the waveform display.

void WaveformDisplay::setRange(juce::Range<double>): This method sets the range of the waveform currently visible in the component.

Conclusion

The WaveformDisplay.h file in the OtoDecks Application application declares the WaveformDisplay class, which represents a component for displaying a waveform of a music track. The class includes private members for the audio thumbnail object and the visible range of the waveform, as well as public methods for loading and drawing the waveform. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

Customize.cpp

The Customize.cpp file contains the implementation of the Customize class, which represents a component for customizing the appearance of the OtoDecks Application application. The Customize class declares various methods for loading and applying custom settings to the application.

Class Members

Private Members

juce::Label titleLabel: This private member represents the label for the title of the customize component.

juce::Label backgroundColorLabel: This private member represents the label for the background color option of the customize component.

juce::TextEditor backgroundColorEditor: This private member represents the text editor for the background color option of the customize component.

juce::TextButton saveButton: This private member represents the button for saving the custom settings in the customize component.

Public Members

Customize::Customize(): This is the constructor for the Customize class. It creates and initializes the customize component and its child components.

void Customize::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the customize component and its child components.

void Customize::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the customize component and its child components.

void Customize::saveSettings(): This method saves the custom settings entered by the user in the customize component.

Conclusion

The Customize.cpp file in the OtoDecks Application application implements the Customize class, which represents a component for customizing the appearance of the application. The class includes private members for the various UI elements in the customize component, as well as public methods for loading and applying custom settings. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

Customize.h

The Customize.h file contains the declaration of the Customize class, which represents a component for customizing the appearance of the OtoDecks Application application. The Customize class declares various methods for loading and applying custom settings to the application.

Class Members

Private Members

juce::Label titleLabel: This private member represents the label for the title of the customize component.

juce::Label backgroundColorLabel: This private member represents the label for the background color option of the customize component.

juce::TextEditor backgroundColorEditor: This private member represents the text editor for the background color option of the customize component.

juce::TextButton saveButton: This private member represents the button for saving the custom settings in the customize component.

Public Members

Customize::Customize(): This is the constructor for the Customize class. It creates and initializes the customize component and its child components.

void Customize::paint(juce::Graphics&): This method overrides the paint() method of the juce::Component class. It is responsible for drawing the customize component and its child components.

void Customize::resized(): This method overrides the resized() method of the juce::Component class. It is responsible for setting the size and position of the customize component and its child components.

void Customize::saveSettings(): This method saves the custom settings entered by the user in the customize component.

Conclusion

The Customize.h file in the OtoDecks Application application declares the Customize class, which represents a component for customizing the appearance of the application. The class includes private members for the various UI elements in the customize component, as well as public methods for loading and applying custom settings. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

DJAudioPlayer.cpp

The DJAudioPlayer.cpp file contains the implementation of the DJAudioPlayer class, which represents an audio player for playing and OtoDecks Applicationpulating music tracks in the OtoDecks Application application. The DJAudioPlayer class implements various methods for loading and playing music tracks, as well as for OtoDecks Applicationpulating their playback speed, volume, and other properties.

Class Members

Private Members

juce::AudioFormatManager formatManager: This private member represents the audio format manager for the DJAudioPlayer object.

juce::AudioTransportSource transportSource: This private member represents the audio transport source for the DJAudioPlayer object.

juce::AudioSourcePlayer sourcePlayer: This private member represents the audio source player for the DJAudioPlayer object.

juce::File audioFile: This private member represents the audio file for the DJAudioPlayer object.

Public Members

DJAudioPlayer::DJAudioPlayer(): This is the constructor for the DJAudioPlayer class. It creates and initializes the audio player object and its child components.

void DJAudioPlayer::loadFile(const juce::File&): This method loads the audio file into the audio player object.

void DJAudioPlayer::play(): This method starts playing the loaded audio file.

void DJAudioPlayer::stop(): This method stops playing the loaded audio file.

void DJAudioPlayer::setSpeed(double): This method sets the playback speed of the audio file.

void DJAudioPlayer::setVolume(float): This method sets the volume level of the audio file.

void DJAudioPlayer::setPosition(double): This method sets the playback position of the audio file.

double DJAudioPlayer::getPosition() const: This method returns the current playback position of the audio file.

Conclusion

The DJAudioPlayer.cpp file in the OtoDecks Application application implements the DJAudioPlayer class, which represents an audio player for playing and OtoDecks Applicationpulating music tracks. The class includes private members for the audio format manager, audio transport source, audio source player, and audio file, as well as public methods for loading, playing, stopping, and OtoDecks Applicationpulating the

audio file. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

DJAudioPlayer.h

The DJAudioPlayer.h file contains the declaration of the DJAudioPlayer class, which represents an audio player for playing and OtoDecks Applicationpulating music tracks in the OtoDecks Application application. The DJAudioPlayer class declares various methods for loading and playing music tracks, as well as for OtoDecks Applicationpulating their playback speed, volume, and other properties.

Class Members

Private Members

juce::AudioFormatManager formatManager: This private member represents the audio format manager for the DJAudioPlayer object.

juce::AudioTransportSource transportSource: This private member represents the audio transport source for the DJAudioPlayer object.

juce::AudioSourcePlayer sourcePlayer: This private member represents the audio source player for the DJAudioPlayer object.

juce::File audioFile: This private member represents the audio file for the DJAudioPlayer object.

Public Members

DJAudioPlayer::DJAudioPlayer(): This is the constructor for the DJAudioPlayer class. It creates and initializes the audio player object and its child components.

void DJAudioPlayer::loadFile(const juce::File&): This method loads the audio file into the audio player object.

void DJAudioPlayer::play(): This method starts playing the loaded audio file.

void DJAudioPlayer::stop(): This method stops playing the loaded audio file.

void DJAudioPlayer::setSpeed(double): This method sets the playback speed of the audio file.

void DJAudioPlayer::setVolume(float): This method sets the volume level of the audio file.

void DJAudioPlayer::setPosition(double): This method sets the playback position of the audio file.

double DJAudioPlayer::getPosition() const: This method returns the current playback position of the audio file.

Conclusion

The DJAudioPlayer.h file in the OtoDecks Application application declares the DJAudioPlayer class, which represents an audio player for playing and OtoDecks Applicationpulating music tracks. The class includes private members for the audio format manager, audio transport source, audio source player, and audio file, as well as public methods for loading, playing, stopping, and OtoDecks Applicationpulating the audio file. Overall, the code is well-organized, easy to read, and follows good practices for naming conventions and commenting.

←----------→

Submitted By: M. Suleman Mirza – 4:29PM – 13/03/2023.